The Atari ST M68000 tutorial part 13 – of hearing that which is spoken

No demo or game is complete without music. The nice blip-blop tunes known as chip music is one of the sweetest forms of music that‘s ever reached my ears. Seems some people don’t quite fancy the type of music the ST has to offer, but I think it’s divine. I’m no musician, that’s probably why a tutorial on sound has been somewhat delayed, but here it finally is. It’s a small tutorial to get down the basics of the sound chip, I intend to follow up with another tutorial on playing the .ym file format, created by Arnaud Carré for the ST-sound project. He’s got a homepage over at http://leonard.oxg.free.fr.

The Atari ST comes equipped with a so called PSG: Programmable Sound Generator. This is yet another chip in the ST, the Yamaha YM-2149, fondly called “yammy”. According to the ST Internals, this cool chip sports lots of features, for example three independently programmable sound generators, 15 logarithmically volume levels and 16 registers. Registers, yes, just give me the tech specs, register addresses and I’ll start to outdo Mozart!

Things aren’t that easy though, it would take insanity to hard code the sound chip. By hard coding I mean just entering numbers into the registers, rather than using some program to make music. There is also a little something here, again according to the ST Internals, it’s not possible to directly address the yammy registers. Instead, you have to put the desired register number in $ff8800, and then you can put data in $ff8802, or read data from $ff8800. Don’t worry, soon comes an explanation of how that applies to real life, but just to be complete, here’s a listing of the registers.

Register	Effect
0,1	Period, length and pitch of channel A
2,3	Period, length and pitch of channel B
4,5	Period, length and pitch of channel C
6	Noise generator
7	Bit 0: channel A tone on/off (0 = on, 1 = off)
	Bit 1: channel B tone on/off (0 = on, 1 = off)
	Bit 2: channel C tone on/off (0 = on, 1 = off)
	Bit 3: channel A noise on/off (0 = on, 1 = off)
	Bit 4: channel B noise on/off (0 = on, 1 = off)
	Bit 5: channel C noise on/off (0 = on, 1 = off)
	Bit 6: port A input/output
	Bit 7: port B input/output
8	Bit 0-3 channel A volume, if bit 4 set, envelope and bit 0-3 ignored
9	Bit 0-3 channel B volume, if bit 4 set, envelope and bit 0-3 ignored
10	Bit 0-3 channel C volume, if bit 4 set, envelope and bit 0-3 ignored
11,12	Sustain, 11 low byte and 12 high byte
13	Waveform envelope
14,15	Port A and Port B, used for output

Like I said, I’m not a musician and I don’t understand to much of this. For me, there are only 4 interesting registers: 7-10. Why? Because with

register 7, I can turn on and off different channels, and with 8-10 I can determine the current volume and also set the volume. In other words, registers 8-10 can be used to fade music in and out, as well as create cool bars that go up and down to the beat of the music (or you can have three pulsating sprites or whatever). Here's some example code on how to use the yammy.

```
        move.b  #7,$ff8800              access Yamaha register 7
        move.b  #%101,$ff8802           turn off channel A and C
```

As the comments say, this will turn off all (tone) sound from channels A and C. Note how only a byte is moved in both instances, and note also that setting a bit means turning the channel off. Here's how to read the volume of channel A:

```
        move.b  #8,$ff8800              channel A volume
        move.b  $ff8800,d0              channel A volume in d0
```

Yep, when you want to read data, you put the register number in $ff8800 as usual, but then you move the data from $ff8800. This obviously means that $ff8800 gets updated between the two move instructions in some way. That's all there is to it actually, but in practice it becomes a little harder.

Sure, we have a basic working of the yammy, well, actually we don't but we know how to use it anyway. Time to play some music perhaps. Most music plays by hooking it up to the VBL, and then just jump to some address in the music file. This of course means that the music file has it's own code routines to play the music, and does not only contain raw music data. The XLR8 chip composor, which can be found at http://dhs.nu , comes with both some example files and the source code for playing them. A good place to start. Here is the code the XLR8 chip composer suggests for playing the music:

```
        pea     0.w
        move.w  #32,-(sp)
        trap    #1
        addq.l  #6,sp

        moveq   #1,d0           ;normal song-play mode
        bsr     music

        move.l  #music+2,$4d2   ;music in VBL
        move.w  #7,-(sp)        ;wait for a key
        trap    #1
        addq.l  #2,sp

        moveq   #0,d0           ;exit music
        bsr     music

        clr.l   $4d2            ;clear VBL
        pea     0.w             ;Back to desktop
```

```
            trap        #1

music       incbin      f:\1.xms                ; musicfile to include
```

Well well, they don't even go out of super visor mode, naughty naughty. Fairly straightforward and easy, there is only one thing that would trouble us, the $4d2 address. We're used to have the VBL hooked up to $70. At address $4ce there are eight long words that point to VBL routines. These VBL routines are executed one after another. So by writing to say $4ce and $4ce+4, we can have two different VBL routines that get executed one after the other.

In the source code above, the author chooses to put the VBL routine in the second of these eight VBL routines. Our way of writing to the $70 instead, is a bit rawer. Writing to $70 disables all VBL routines except the $70. This means that we know that our, and only our VBL routine is the one to run. In any way, if we spot a memory address close to $4ce in some future source code, we know that it's a VBL routine. Translated into how we would code it, it looks like this:

```
            jsr         initialise

            moveq       #1,d0                   normal song play
            bsr         music                   start music

            move.l      $70,-(a7)               backup $70
            move.l      #main,$70               main routine on VBL
            move.w      #7,-(a7)
            trap        #1
            addq.l      #2,a7                   wait keypress
            move.l      (a7)+,$70               restore $70
            jsr         restore

            moveq       #0,d0
            bsr         music                   stop music

            clr.l       -(a7)
            trap        #1                      exit

main
            movem.l     d0-d7/a0-a6,-(a7)       backup registers

            bsr         music+2                 play music

            movem.l     (a7)+,d0-d7/a0-a6       restore registers
            rte

            include     initlib.s

            section data
music       incbin      1.xms       musicfile to include
```

Well, what do you know, ours became longer, but it's built for more add-ons, and it also has some backup feature such as getting out of supervisor mode, and it also has a section data. Doesn't really matter, both ways are equally fast really. Speaking of fast, there's an instruction here that I don't think we've encountered before, the moveq instruction. Moveq stands for MOVEQuick. It works pretty much as a normal move, but it can only move quantities in the range of -128 to +127 (a byte). The data does get sign extended though, meaning it will take up a 32-bit quantity (long word). Thus a moveq.l #0,d0 clears d0 faster than a clr.l d0. Handy little instruction actually. You will also notice how I put $70 on the stack instead of saving it to a variable.

So we have a way of playing music, at least music composed with the XLR8 chip composor. This is a little thin, so against our better knowledge, we decide it would be fun to do some volume meters as well. In order to do these VU bars, we have to play the music, read registers 8-10 (for volume) and then paint the VU bars. Yes, it's true, the yammy has three sound channels, meaning it can play up to three different sounds at once. It also has some noise generator I think so it's able to play four different sounds at once.

The volume is represented by the four least significant bits, meaning it's a value between 0 and 15 (%1111), would be smart to paint one line of VU bar for each volume, right? So at volume 15, the VU bar takes up 15 lines, this can be a little small though, so just for fun we decide to leave every second line blank (background coloured), thus volume 15 will take up 30 lines instead. Since we have 15 different VU lines, each line can have it's own colour and we'll still have one over for the background as well. Seems the ST was made for these things! Actually, it wasn't, it's just normal for computers to have many things on binary bounders. Thus the powers of two (such as 16, or 0-15) show up a lot.

As we know, the volume data may contain other stuff than just the four volume bits, so in order to keep only those bits, we have to and off the other bits. Otherwise the volume data might contain a number larger than 15 and that will screw us up big time, making us do stupid things like drawing outside of the screen, which will probably result in a crash.

Now, to decide on how to draw the volume bars. What, should this be a problem? Just do a dbf loop according to the volume and paint as many lines as the volume. Yes, that won't work. Say one VBL the volume is 12, then the other VBL it's 5, but the VU bar will still be 12 lines high since we don't delete it.

So on every VBL, we first delete the VU bar and then paint it. This can be done, but actually it's smarter to first paint the VU bar, then delete it. The delete part is more generic, and thus easier to fit into a loop, while the paint part requires colour updates and so on. Thus, the VU bar routine will be to first paint all three VU bars to the max, then delete as many lines as the inverted volume (volume 15 means delete nothing, volume 0 means delete 15). This will work nicely. Actually, theory part over, time for source code:

```
jsr         initialise

moveq       #1,d0                   normal song play mode
bsr         music
```

```
            move.l      #palette,a0                 pointer to palette
            movem.l     (a0)+,d0-d7                 palette in d0-d7
            movem.l     d0-d7,$ff8240       apply palette

            move.l      $70,-(a7)           backup $70
            move.l      #main,$70           start main routine

            move.w      #7,-(a7)
            trap        #1
            addq.l      #2,a7               wait keypress

            move.l      (a7)+,$70           restore old $70

            moveq       #0,d0               stop music
            bsr         music

            jsr         restore

            clr.l       -(a7)
            trap        #1                  exit

main
            movem.l     d0-d7/a0-a6,-(a7)

            bsr         music+2             play music

* put in VU meters
            move.l      $44e,a0             get screen address
            add.l       #160*199-(15*2)*160,a0      bottom area of screen
            move.l      #meter,a1           point to meter colours

            rept        15                  15 max volume
            movem.l     (a1)+,d0-d1                 VU meter colour in
d1-d2
            movem.l     d0-d1,(a0)          first VU meter
            addq.l      #8,a0               next VU meter
            movem.l     d0-d1,(a0)          second VU meter
            addq.l      #8,a0               next VU meter
            movem.l     d0-d1,(a0)          third VU meter
            add.w       #320-16,a0                  two lines down, two
meter left
            endr

* delete VU meters depending on volume
            move.l      $44e,a0             get screen address
            add.l       #160*199-(15*2)*160,a0      bottom area of screen

            moveq.l     #0,d0               clear d0
            move.b      #8,$ff8800                  chanenl a volume
```

```
            move.b      $ff8800,d0                  put volume in d0
            jsr         del_meter

            moveq.l     #0,d0               clear d0
            move.b      #9,$ff8800                  channel b volume
            move.b      $ff8800,d0                  put volume in d0
            add.l       #8,a0               next VU meter
            jsr         del_meter

            moveq.l     #0,d0               clear d0
            move.b      #10,$ff8800                 channel c volume
            move.b      $ff8800,d0                  put volume in d0
            add.l       #8,a0               next VU meter
            jsr         del_meter

            movem.l     (a7)+,d0- d7/a0- a6
            rte


del_meter
* screen address of top line in a0
* volume in d0, gets detroyed
            move.l      a0,- (a7)           backup a0
            move.l      a1,- (a7)           backup a1
            and.b       #%1111,d0           keep only lowest 4 bits

            move.l      #delete,a1          beginning of delete blocks
            mulu        #12,d0              length of one delete block
            add.l       d0,a1               skip some delete instructions
            jmp         (a1)                jump to correct delete position

delete
            rept        15
            clr.l       (a0)                clear two bit planes
            clr.l       4(a0)               clear two bit planes
            add.l       #320,a0             hop two lines down
            endr

            move.l      (a7)+,a1            restore a1
            move.l      (a7)+,a0            restore a0
            rts


            include     initlib.s


            section data

meter
* colour data for each line of VU meter
```

```
            dc.w    $00ff,$00ff,$00ff,$00ff
            dc.w    $0000,$00ff,$00ff,$00ff
            dc.w    $00ff,$0000,$00ff,$00ff
            dc.w    $0000,$0000,$00ff,$00ff
            dc.w    $00ff,$00ff,$0000,$00ff
            dc.w    $0000,$00ff,$0000,$00ff
            dc.w    $00ff,$0000,$0000,$00ff
            dc.w    $0000,$0000,$0000,$00ff
            dc.w    $00ff,$00ff,$00ff,$0000
            dc.w    $0000,$00ff,$00ff,$0000
            dc.w    $00ff,$0000,$00ff,$0000
            dc.w    $0000,$0000,$00ff,$0000
            dc.w    $00ff,$00ff,$0000,$0000
            dc.w    $0000,$00ff,$0000,$0000
            dc.w    $00ff,$0000,$0000,$0000
            dc.w    $00ff,$0000,$0000,$0000

palette
            dc.w    $000,$093,$09b,$094,$09c,$095,$09d,$096
            dc.w    $09e,$097,$29f,$39f,$49f,$59f,$69f,$79f

music       incbin  instinct.xms                    musicfile
```

Here we go! Setup the music to work, then put in the palette. Backup the old $70 by putting it on the stack, put in my own $70 routine, wait for keypress, shut down music, restore and exit. All the usual stuff.

The main routine starts off by playing the music, that one simple command is enough to keep the music running. Now comes the interesting part: putting in the VU bars. I get the screen address, and go to the bottom area of the screen. This means go to the absolute bottom, line 199, and then hop 30 lines up. Because I want to paint the entire VU bar, and the VU bar is 15*2 lines high (max 15 volume and every second line is interlaced).

The actual painting of the VU bar is a bit tricky. I point to the bar label, which contains colour data for the VU bars. Each little four word block here is data for a bit plane, so the first line is colour 15. The reason for the two leading 0's, is that I don't want the entire bit plane filled, in this way, only 8 pixels out of sixteen will be set.

```
            dc.w    $00ff,$00ff,$00ff,$00ff
```

Is the same as

```
            dc.w    %0000000011111111
            dc.w    %0000000011111111
            dc.w    %0000000011111111
            dc.w    %0000000011111111
```

And we know that by putting this into the screen memory, we will have 8 pixels with colour 0, and then 8 pixels with colour 15. The next entry is

```
dc.w    $0000,$00ff,$00ff,$00ff
```

Which is the same as

```
dc.w    %0000000000000000
dc.w    %0000000011111111
dc.w    %0000000011111111
dc.w    %0000000011111111
```

When we put this into screen memory, we get colour 14 in the last 8 pixels.

When pointing to the bar label, a1 points to memory that contains this data: $00ff00ff00ff00ff. This data I move into d0 and d1 with a movem instruction, then I put that data into the screen memory. Adding 8 to the screen memory pointer will put me on the next VU bar, 16 pixels to the right, and then I move that same colour data into the screen memory there, and repeat one last time. Then I need to correct the screen pointer: by adding 320, I move two lines down, and then I need to subtract 16 from that to be on the first VU bar position. Repeat all this 15 times to paint in all three VU bars full.

Now the time has come to delete the VU bars, so that they will reflect the value of the volume. Again, get screen memory and go to the bottom area, pointing right at the topmost line of the first VU bar. Clear d0 just to be sure there's no garbage, and read Yamaha register 8, which is channel A volume. Now the volume is in d0, and the screen address is in a0, jump into the del_bar routine to delete the bar.

The del_bar routine is also a bit tricky, and uses an almost dirty method. Backup the registers so that they don't get destroyed. This is good practice for sub routines, so that other programmers can count on calling routines without having data destroyed. And away all bits but the first four. Now we have pure volume data in d0.

It would be tempting to just go through a dbf loop to clear out the lines, but this won't work. A bdf loop always executes once, but in some circumstances, we don't want the delete loop to execute even once. So instead of having a loop, I have 15 blocks of delete data, each block deletes one line of VU bar. By jumping into the correct block, I take away the exact number of VU bar lines. Each delete block looks like this:

```
clr.l   (a0)            clear two bit planes
clr.l   4(a0)           clear two bit planes
add.l   #320,a0         hop two lines down
```

This will clear out the four bit planes of a line, and then hop two lines down. This block takes 12 memory positions. Usually, an instruction takes a long word to store, these three instructions are no exceptions. Since all instructions just get loaded into memory, we can easily jump to them. Go into MonST mode to see this clear, in the instruction window, you'll see all instructions, and to the left of them you'll see what memory position they occupy. By adding 4 to the program counter, you usually skip one instruction.

For example, by jumping to the start of the delete blocks+12, we will skip one delete block. In the del_bar routine, I have 15 delete blocks, I let a1 point to the beginning of these blocks. Then I multiply the volume by 12, since this is the size of each delete block, add that value to a1, and jump to the address a1 contains.

Say that we have volume one, this means execute 14 delete blocks, which will leave only one line of VU bar left. 1*12 = 12, thus we will jump to the beginning of the delete blocks+12, which will let us skip one delete block, and then we have 14 left. Here's how it looks:

```
Memory position (fictional)
$0      move.l    $10,a1        beginning of delete blocks
$4      mulu      #12,d0        length of one delete block
                                d0 contains 12
$8      add.l     d0,a1         skip some delete instructions
                                a1 now contains $1c
$c      jmp       (a1)          jump to $1c

$10     clr.l     (a0)          clear two bit planes
$14     clr.l     4(a0)         clear two bit planes
$18     add.l     #320,a0       hop two lines down

$1c     clr.l     (a0)          clear two bit planes
$20     clr.l     4(a0)         clear two bit planes
$24     add.l     #320,a0       hop two lines down

$28     clr.l     (a0)          clear two bit planes
$2c     clr.l     4(a0)         clear two bit planes
$30     add.l     #320,a0       hop two lines down
        ...                     13 more delete blocks
```

That's that. In short, the program only runs a VBL routine. This VBL routine plays the music, and then paints in VU bars at max. Then the volume is read from yammy registers, for each volume read, the del_bar routine is called which deletes as many lines as the inverted volume. Then add 8 to the screen memory to point to the next VU bar, read the volume and call the del_bar routine.

With this knowledge of the volume workings, you can have just about any effect tied to the volume. I first had the background colour be set by the three channels, channel A for red colour, channel B for green and channel C for blue. This created quite the psychedelic background I can tell you :) One cool thing would be to have three Xenon 1 ships, that morph back and forth between tank and ship, say that volume 15 means complete tank morph, and 0 means ship, then volume 7 would be morph in between those. Once again, your fantasy can run free!

In the next tutorial, I hope to cover the .ym file format as described in the beginning. This will mean setting up our own routine to write raw data into the sound chip, which should be quite easy. Just put register number as usual, then write the data found in the .ym file. Stay tuned …

perihelion of poSTmortem, 2002-07-28

“I can kill with a word.”
- Dune, the Movie